An Efficient Implementation of Tiled Polymorphic Temporal Media

Simon Archipoff

An Efficient Implementation of Tiled Polymorphic Temporal Media

Simon Archipoff *
simon.archipoff@u-bordeaux.fr
LaBRI
Université de Bordeaux

Abstract

Tiled Polymorphic Temporal Media (TPTM) are a convenient way to describe, compose and encode multimedia streams. This paper presents a TPTM encoding that allows simple and efficient implementation of both composition and rendering. In particular, an on-the-fly rendering procedure is provided in order to handle infinite (lazy) TPTM.

1 Introduction

In the context of computerized music, but also in the context of systems producing temporal media, the purpose of computation not only concerns what values are to be computed as in classical programming, but also when these values are to be produced or rendered.

In these fields, temporal media (sequences of media values that evolve with time, such as audio or video) play a prominent role. In [2], a polymorphic temporal media algebra is defined, allowing to combine them either in sequence or in parallel. Presumably, these operators are sufficient. However, the analysis of common functions, such as temporal media synchronization, suggests that temporal media may be abstracted further.

In order to compose of two audio sequences m_1 and m_2, we have to take into account the content of m_1 and m_2 in order to make a consistent audio alignment. A silence d may have to be added to the composition in order to synchronize m_1 and m_2. More precisely, such a composition would be $(d :+: m_1) :=: m_2$ or $m_1 :=: (d :+: m_2)$, where $:+:$ denotes the sequential composition and $:=:$ denotes the parallel composition. Of course, an abstraction of the form

$$\lambda m_1\ m_2 \to (d :+: m_1) :=: m_2$$

would be meaningless since the value d depends on both m_1 and m_2. Instead, the correct abstraction of the composition of two audio sequences m_1 and m_2 can be defined by:

$$\begin{aligned}\lambda m_1\ m_2 \to\ & \textbf{let}\ d = \mathit{alignOffset}\ m_1\ m_2 \\ & \textbf{in if}\ d > 0 \\ & \qquad \textbf{then}\ (d :+: m_1) :=: m_2 \\ & \qquad \textbf{else}\ m_1 :=: ((-d) :+: m_2)\end{aligned}$$

where the function *alignOffset* performs a possibly complex analysis of the underlying musical structure induced by m_1 and m_2.

The synchronization data of two temporal media m_1 and m_2 can be independent one with the other, for instance for musical bars. This suggests that the underlying abstract structure that may relevantly describes these audio sequences should not only consists in the ordered audio frames themselves, that form the basis of the audio sequences, but also in a number of synchronization marks al-

*Partially funded by INEDIT, ANR-12-CORD-009.

lowing to adequately align these audio sequences one with each other.

As a matter of fact, most audio toolboxes that are available nowadays propose mechanisms to enrich audio or musical sequences with such a kind of synchronization markers. Yet, while these markers are defined (in a quite ad hoc way) at the interface level, their definition, manipulation and transformation at the programmatic level is far from being clearly understood.

Despite an apparent simplicity, defining and handling adequate synchronization marks over temporal media is not easy: how many synchronization marks should we allow? This immediately raises the question of representing the (possibly many and heterogeneous) time scales the temporal media are to be defined on. Such a time scale can be for instance: seconds, bars, beats, events... Even more, as soon as interaction appears, this poses the problem of building systems that are globally asynchronous (event driven) and locally synchronous (almost continuous).

The tiled modeling and tiled programming approach that is recalled and implemented in this paper is an attempt to formalize, not only at the programmatic level but also at the modeling level, the interplay between temporal media values: what they are, and their synchronization marks: how they should be used. In particular, it aims at providing sufficiently rich abstract description of temporal media types so that as we shall see, one can easily define a meaningful abstract composition process:

$$\lambda m_1\ m_2 \rightarrow m_1\ \%\ m_2$$

where % is the tiled composition operator. The resulting domain specific language is both versatile and efficient.

In a first part of this paper we will recall some of the work done in this field. Then we will present our main contribution: a new implementation of Tiled Polymorphic Temporal Media. Last, we will study how this implementation compare to other ones considering algorithmic complexity, performances, and finally expressiveness. As an illustration of expressiveness, we present a set of primitives to convert Polymorphic temporal media to Tiled Polymorphic Temporal Media using our implementation.

2 From PTM to TPTM

In this section we give a brief survey of the existing approaches, from Hudak's Polymorphic Temporal Media (PTM) to the more recently proposed Tiled Polymorphic Temporal Media (TPTM).

An important concept with PTM and therefore TPTM is the notion of observational equivalence between two temporal media. The observational equivalence of m_1 and m_2, denoted by $m_1 \equiv m_2$, means that m_1 and m_2 can not be distinguished during the rendering, no matter how they are combined with other temporal media.

2.1 Polymorphic Temporal Media (PTM)

A Hudak's Polymorphic Temporal Media [2] can be either a temporal media value lasting for some duration d, a sequential composition :+: or a parallel composition :=: of any two simpler media. Such operators are depicted in Figure 1 where, as in most figures, the horizontal axis represents the flow of time from left to right.

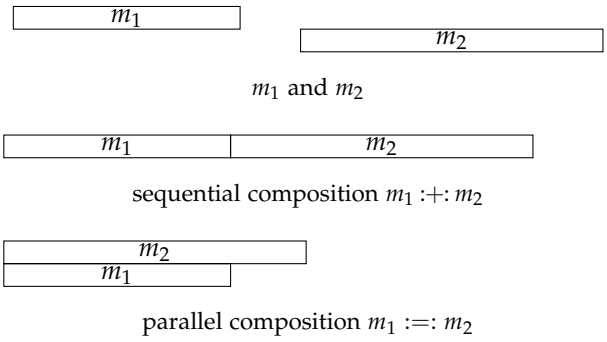


Figure 1: PTM syntax and semantics

Despite its simplicity, this model is used in computer music library such as Haskore [6] and Eu-

terpea [3] (both written in Haskell). It has nice algebraic properties [2].

This model handles infinite temporal media thanks to laziness. For instance in $m = m_1 :+: m$, m is a valid and playable temporal media when m_1 is finite. About the implementation, the "on demand" semantics of Haskell spares the programmer a lot of work and makes the code very simple.

With the simplicity of this approach come two slight drawbacks. The first one concerns the semantics of the parallel composition, we have many ways to synchronize two temporal media in parallel (at their beginning, at their end, truncating the end of the longest...). We have to make an arbitrary choice: in Euterpea, the parallel composition synchronizes the two operands at their beginning, and the duration of the resulting media is the one of the longest operand. The second drawback is the lack of modularity mentioned in the introduction.

2.2 Tiled Polymorphic Temporal Media (TPTM)

Tiled Polymorphic Temporal Media proposed by Hudak and Janin [4] offers a simpler approach with a single tiled composition operator (denoted %). With TPTM, the information about the synchronization is embedded in the media and is used in the tiled composition. It follows that both sequential and parallel compositions appear as particular cases of tiled composition.

Tiled PTM allows the decorrelation of the beginning and the end of the temporal media from their synchronization points. More precisely, a tile is a stream enriched with two particular points called pre ($\curlyvee$) and post ($\downarrow$), as depicted in Figure 2.

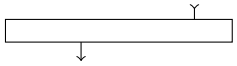
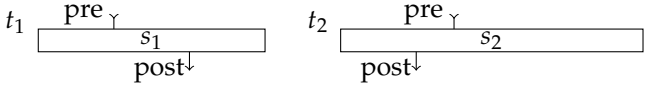
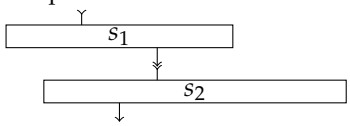


Figure 2: A tile

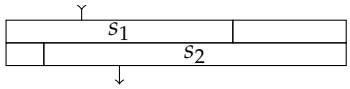
Then, given two tiles t_1 and t_2 encapsulating streams s_1 and s_2 respectively, we compute the composition $t_1 \,\%\, t_2$ by synchronizing the post marker of t_1 with the pre marker of t_2, and mixing their underlying streams. The synchronization points of the resulting tile are the pre marker of t_1 and the post marker of t_2, as depicted in Figure 3. Thus, tiles t_1 and t_2 describe how s_1 and s_2 are aligned.



(a) Two stream descriptors enriched with pre and post synchronization points



(b) When composing t_1 with t_2, the post synchronization point of t_1 is aligned with the pre synchronization point of t_2



(c) The streams s_1 and s_2 are completed with silences and composed together to make a new tile

Figure 3: Tiles are enriched streams

2.3 Derived Operators

The model is rich enough so that we can easily define derived functions to compose tiles. The *reset* of a tile *re t* is a tile of duration 0 with the post marker at the same place as the pre marker in *t*. It can be used to make parallel composition of tiles by synchronizing them at their pre marker. The *coreset* function *co* places the pre marker of a tile at the position of the post marker, making a tile of duration 0. It is useful to synchronize two parallel tiles on their post marker. And the *inv* function swaps the pre and the post marker. These functions are depicted in Figure 4.

The *inv* function have a very interesting algebraic property. Under a few hypothesis on the underlying media, the set of tiles equipped with product % and the empty tile *unit* (of duration 0) forms an inverse monoid with regard to observational equiva-

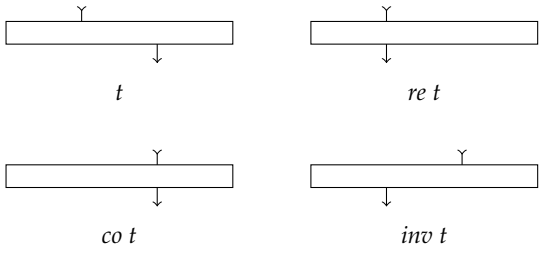


Figure 4: A tile t, and its reset, co-reset, and inverse

lence [4]. More precisely the product % is associative, *unit* is neutral and for every tile x there is a unique tile *inv* x verifying:

$$x \,\%\, inv\ x \,\%\, x \equiv x$$
$$inv\ x \,\%\, x \,\%\, inv\ x \equiv inv\ x$$

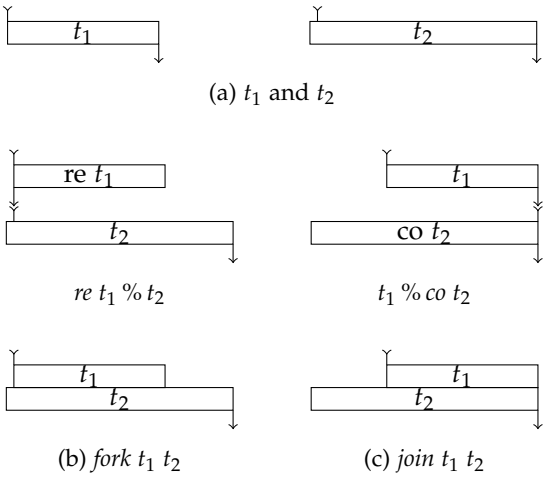


Figure 5: The *fork* and *join* functions

2.4 Parallel Operators

Using those operators, we can define *fork* and *join* binary operators, two parallel compositions synchronizing tiles at their pre and post marker, respectively. The *fork* operator behaves somehow like the :=: compositor of PTM. The semantics of these functions are depicted in Figure 5.

$$fork\ t_1\ t_2 = re\ t_1 \,\%\, t_2$$
$$join\ t_1\ t_2 = t_1 \,\%\, co\ t_2$$

2.5 Modeling Examples

Tiled Polymorphic Temporal Media are convenient to handle media according to their underlying structures beginning or ending, which can be distinct from their literal beginning or ending. Possible applications of this property are detailed below.

2.5.1 Adding a Pickup

In music, an anacrusis, or pickup, is a brief introduction to a measure. Using TPTM, it is simple to add a pickup to a tile. Given a tile t with a positive duration and nothing before the pre marker, it is possible to add a pickup p by computing $co\ p \,\%\, t$ as depicted in Figure 6.

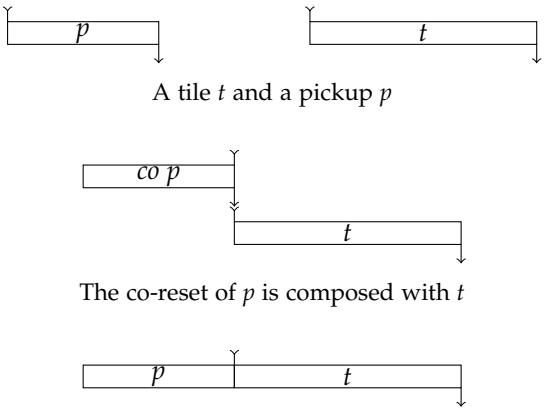


Figure 6: Adding a pickup with tiles

In this example, we can see a case where TPTM are more modular than PTM. In $a \,\%\, t \,\%\, b$, if one wants to add something in t, as long as the markers of t are placed in a consistent way, neither a nor b will be impacted by this modification.

2.5.2 Uniform Handling of Language and Rhythmical Units

As a more concrete application instance, tiles are convenient to modelize the relation between the lyrics and the music. Those two objects do not have the same logical structure. One is decomposed in verses, grammatical units, words,... the other in phrases, bars,... Nevertheless, they are both entangled in a song and TPTM can formalize these relations.

Here, we consider the Bob Dylan's song "Blowing in the wind". We can encode the lyrics by a list of tiles where the text embedded by each tile is determined by grammar, and the markers of those tiles by musical criteria. In Figure 7 the lyrics are split using grammatical criteria, the pre and post markers are placed according to the musical bars location.

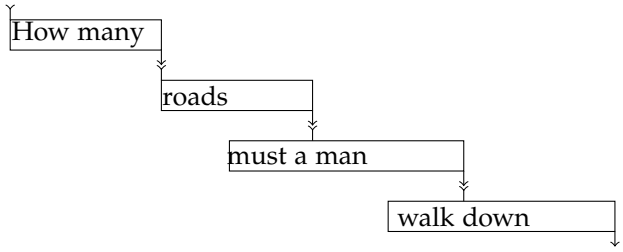


Figure 7: A song tiled by measures

We build a list of tiles with a duration of one measure. By packing these tiles four by four, we tile the verses of the song, as shown in Figure 8.

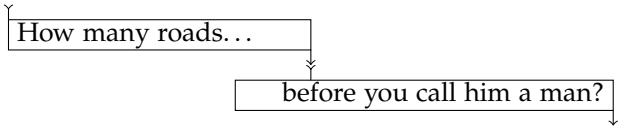


Figure 8: A song Tiled by verses of 4 measures

This illustrates the potential of TPTM to represent music at various time scales, and according to the different structures involved (natural language, measure, melody,...).

3 Existing Implementations

Implementing TPTM is not a trivial task. We have to satisfy the constraint of polymorphism, this implies to make minimal hypothesis on the tiled media. We also want to handle infinite data structure, and we want our tiles to be playable in real time.

3.1 Tiles in Euterpea

Hudak and Janin made a first implementation on top of Euterpea [4] by encapsulating Euterpea's *Music* type into a type *Tile*:

$$\begin{aligned}\textbf{data}\ \mathit{Tile}\ a = \mathit{Tile}\ \{&\mathit{preT}\ :: \mathit{Duration};\\ &\mathit{postT} :: \mathit{Duration};\\ &\mathit{musT} :: \mathit{Music}\ a\}\end{aligned}$$

with *Duration* implementing the set of rational numbers.

Simple and elegant, this implementation already allows to study the notion of tiled programming. However, it suffers from a lack of polymorphism, as primitives on tiles are implemented with primitives of *Music* type.

3.2 The Syntactic Approach

Other standalone implementations proposed by Bazin, Hudak and Janin [1, 5] separate clearly the out-of-time tile construction induced by tiled modeling and the in-time on-the-fly rendering. To this end, they encode tiles in a syntactic way.

This settles a set of primitives that allows both composition and rendering of tiles.

3.3 Creation Primitives

First, the out-of-time primitives used to produce tiles:

$$\begin{aligned}&\mathit{delay} :: \mathit{Duration} \to \mathit{Tile}\ a\\ &\mathit{event} :: a \to \mathit{Tile}\ a\\ &(\%)\ \ :: \mathit{Tile}\ a \to \mathit{Tile}\ a \to \mathit{Tile}\ a\\ &\mathit{dur}\ \ :: \mathit{Tile}\ a \to \mathit{Duration}\end{aligned}$$

This syntax describes the two atomic tiles. The expression *delay d* is an empty tile of duration *d* which can be positive as well as negative, and *event e* is a tile of null duration with a single event *e*. The Figure 9 shows two atomic tiles. In most figures the horizontal axis represent the time and the scale is respected. The *dur* function returns the duration of the tile, defined as the time distance between the pre and post markers. We observe that $dur\ (a \,\%\, b) \equiv dur\ a + dur\ b$. The % operator allows the composition of those two base tiles.

(a) The tile *delay* 2 (b) The tile *event e*

Figure 9: Two simple tile

As delays can be negative, the temporality of the media does not have to be reflected in the description of the tile. For instance, in $event\ a \,\%\, delay\ (-1) \,\%\, event\ b$, the event *b* will occur before the event *a*. Then, we can describe every tile by its sequence of events separated by back and forth delays, hence the resulting tiles are "zigzag" through time. An example of such a tile is depicted in Figure 10.

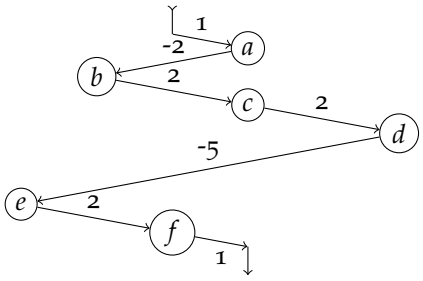


Figure 10: A tile "zigzag" *t*

With those construction primitives, we can implement the reset, co-reset, and inverse functions described earlier. These functions are depicted in Figure 4.

$$
\begin{aligned}
&inv\ t = \textbf{let}\ d = dur\ t \\
&\qquad \textbf{in}\ delay\ (-d) \,\%\, t \,\%\, delay\ (-d) \\
&re\ t = t \,\%\, delay\ (-(dur\ t)) \\
&co\ t = delay\ (-(dur\ t)) \,\%\, t
\end{aligned}
$$

3.4 Rendering Primitives

Obviously, zigzag tiles can not be rendered as they are. The first event to occur can be deep in the syntactic structure. For each tile, there is a normal form where there is no consecutive delays, and no negative delay between two events. Such a normal form is depicted in Figure 11.

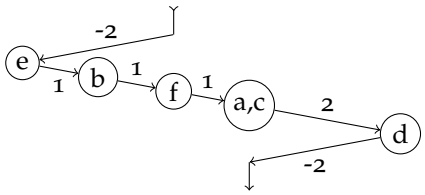


Figure 11: The tile *t* normalized

One way to compute such a normal form is to define some primitives to extract the first events and the time distance to them:

$$
\begin{aligned}
&firstE :: Tile\ a \to MultiSet\ a \\
&firstD :: Tile\ a \to Maybe\ Duration
\end{aligned}
$$

The *firstE* function returns the list (encoding a multiset) of all the first events (all the other are strictly later). And the *firstD* function returns the delay between the pre marker and the first events. Such a delay will not be defined if the tile have no event. That is the reason why *firstD* returns a value of type *Maybe Duration*. It can be either *Just d*, returned when the tile have events, or *Nothing* when there is no such event.

In order to play a tile, we need to compute the first events to occur, when they will occur, but also what comes afterward. We define the primitives *headT* and *tailT* to compute these. The *headT* function returns the tile composed of the first delay and the first simultaneous events, and *tailT* returns the remainder of the tile, as depicted in Figure 12.

```
headT :: Tile a → Tile a
tailT  :: Tile a → Tile a
```

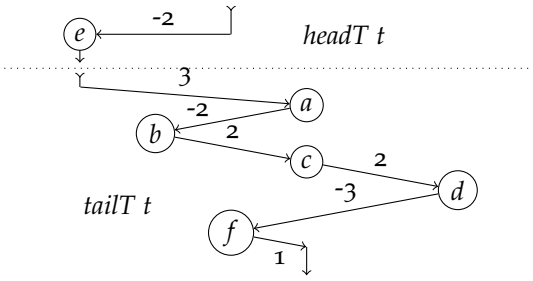


Figure 12: Same tile as in Figure 10 with the first event extracted

For any tile, we have the property:

$$t \equiv headT\ t \,\%\, tailT\ t$$

Those functions are named after the *head* and *tail* functions defined over lists because of the semantic similarity between them. Over any non empty list l the following property is verified.

$$l == head\ l : tail\ l$$

The *headT* function can be defined in term of *firstD* and *firstE*:

```
headT t =
  case firstD t of
    Nothing → delay 0
    Just d  → delay d % fold (λt e → t % event e)
                             (delay 0)
                             (firstE t)
```

As *event* only takes a single event, the multiset returned by *firstE* is reduced in a tile using tiled product and the standard Haskell function *fold*. This function takes as parameter an accumulation function (here of type $Tile\ e \to e \to Tile\ e$) and a starting accumulator value (here *delay* 0, a neutral).

3.5 Observational Equivalence

The simplest observational equivalence is the following: two tiles are considered equivalent when they have the same duration and describe the same marked sequence of timed events. Such a definition does not takes into account the properties of the events type, relevant from the observation. It is recursively implemented by:

```
(≡) :: Eq e ⇒ Tile e → Tile e → Bool
(≡) t1 t2 =
  case (firstE t1, firstE t2) of
    ([], [])    → dur t1 == dur t2
    (le1, le2) → firstE t1 == firstE t2
                 ∧ firstD t1 == firstD t2
                 ∧ tailT t1 ≡ tailT t2
```

The base case is when both tiles have no event, then it checks if they are equal in duration. The recursive case checks if the two tiles have the same first events, at the same time, and if their remainders (their tails) are equivalent.

4 Heaped Tiles

Here, we present a new approach using balanced data structure and making the duration of the tile independent of the delays used to build it. For this, we define a tile as a duration (of type *Duration*) and a set of timed events. Such a set is implemented by a mergeable heap, here Sleator and Tarjan's skew heap [7].

The implementation of the tile is given by:

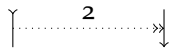
```
data Tile e = Tile Duration (SHeap e)
```

with *Duration* the duration of the tile and *SHeap e* a set of timed events.

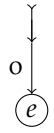
The skew heap of events is a binary tree where node are labeled by a delay and embed some events.

```
data SHeap e = Empty
             | SHeap Duration (MultiSet e) (SHeap e)
                              (SHeap e)
```

The events are stored in a multiset *MultiSet*, the *Duration* member is the time distance that separates the events in the node from his father, or the implicit pre marker of the tile for the root. In Figure 13 are depicted the same tiles as in Figure 9. The duration is represented by a dotted line between the two synchronization points, and the event is attached to the implicit pre marker of the tile (at time 0).



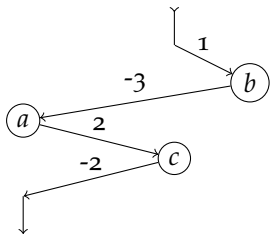
(a) The tile *delay* 2



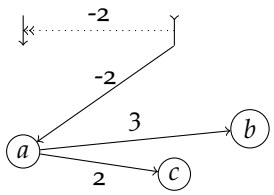
(b) The tile *event e*

Figure 13: Two simple heaped tiles

In Figure 14, we can see the correspondence between zigzag and heaped representations (same position on the horizontal axis means same time). In the later case, the syntactic composition of the tile is lost but the events are partially ordered.



(a) Zigzag



(b) Heaped

Figure 14: Two representations of the tile *delay* 1 % *event b* % *delay* (-3) % *event a* % *delay* 2 % *event c* % *delay* (-2)

4.1 High Level Implementation of the Primitives

The primitives introduced in 3.3 are implemented as follows:

```
delay d = Tile d Empty
event e = Tile 0 (SHeap 0 (singleton e) Empty Empty)
      -- singleton e is a multiset containing only e
(%) (Tile d1 s1) (Tile d2 s2) =
   Tile (d1 + d2) (mergeSH s1 (shiftSH d1 s2))
dur (Tile d _) = d
```

The *delay d* function returns a tile of duration d with no events, the *event e* function returns a tile of duration 0 and a single event at time distance 0 of the pre marker of the tile.
The *mergeSH* function is a binary operator that merges two heaps by preserving time distances, every event of each operand will be at the same time distance in the resulting tree. And the *shiftSH* function takes as parameter a duration and a heap, and shift all the timed events of the heap by the duration.
The implementation of composition is based on the property:

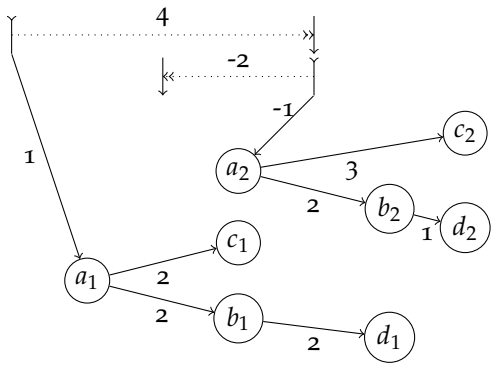
$$t_1 \,\%\, t_2 \equiv re\ t_1 \,\%\, delay\ (dur\ t_1) \,\%\, t_2$$

The sole SHeap of the tile *delay* $(dur\ t_1) \,\%\, t_2$, is computed by *shiftSH* d_1 s_2 (every events of t_2 are preceded by a delay of $(dur\ t_1)$). Then, the events of t_2 shifted are merged by the ones of t_1, and the duration of the resulting tile is the sum of the durations of t_1 and t_2. The tiled product is depicted in Figure 15.

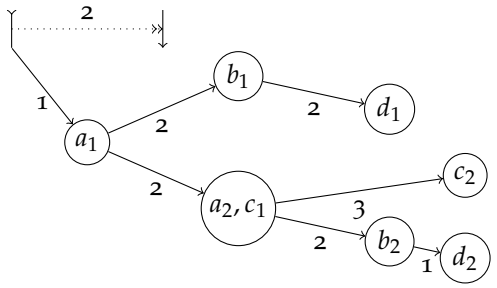
4.2 Additional Primitives

The rendering primitive are simple as well, all of the relative complexity of the implementation is hidden in the heap's primitives. Here the implementation of a few more primitives:

```
firstD (Tile _ (SHeap d _ _ _)) = Just d
firstD _ = Nothing
```



(a) The synchronization points are aligned (the heap of the second operand is shifted)



(b) The heaps and the synchronization data are merged

Figure 15: Tile composition

```
firstE (Tile _ Empty) = empty    -- empty multiset
firstE (Tile _ (SHeap _ e _ _)) = e
headT (Tile _ Empty) =
      Tile 0 Empty
headT (Tile _ (SHeap t e _ _)) =
      Tile t (SHeap t e Empty Empty)
tailT (Tile d Empty) =
      Tile d Empty
tailT (Tile d (SHeap t _ l r)) =
      Tile (d − t) (mergeSH l r)
```

The *tailT* function removes the first events, and then computes a new heap containing all the remaining events.

A very important improvement with this tile representation is that the reset of a tile is defined even when the duration of this tile is undefined, as we can implement it by:

$$re\ (Tile\ \_\ h) = Tile\ 0\ h$$

4.3 Low Level Implementation

A skew heap [7] is a very simple data structure that can be implemented in Haskell with less than 10 lines of code, it have nice algorithmic properties, and as it is a good base for an implementation of a priority queue, we can use it to implement TPTM as well. It is a binary tree where the nodes are labeled by elements of an ordered set (times for instance). It allows a quick access to the smallest element of the heap, and a quick merge of two heaps. Here, to store the events we uses a slightly modified skew heaps, the difference with the usual ones is that the nodes carry delays and not absolutes times.

The set of timed event is coded by a binary tree. In an expression of the form

$$n = SHeap\ d\ mse\ lchild\ rchild$$

where d is the time distance between the events in the multiset *mse* and the event embedded by the father of n, or the implicit pre synchronization of the tile when n is the root.

The invariant of the heap is:

- the duration carried by every node except the root is strictly positive,
- a node carry at least one event,
- the total delay to an event is the sum of all delays from the root to the node carrying this event.

The implementation of *shiftSH* is pretty straightforward. It is an $\mathcal{O}(1)$ operation: as the labels on the nodes are relatives, shifting a node shifts all its subtree. With absolutes labels we would have an $\mathcal{O}(n)$ operation.

```
shiftSH :: Duration → Sheap e → SHeap e
shiftSH d Empty = Empty
shiftSH d (SHeap d' e l r) = SHeap (d + d') e l r
```

Finally, the implementation of *mergeSH* is given by:

```
mergeSH :: Ord e ⇒ SHeap e → SHeap e → SHeap e
mergeSH l Empty = l
mergeSH Empty r = r
mergeSH heap1@(SHeap d1 e1 l1 r1)
        heap2@(SHeap d2 e2 l2 r2) =
  case compare d1 d2 of
    EQ → SHeap d1 (union e1 e2) (mergeSH l1 r1)
                                (mergeSH l2 r2)
    LT →
      SHeap d1 e1 (mergeSH r1 (shiftSH (−d1) heap2))
                  l1
    GT →
      SHeap d2 e2 (mergeSH r2 (shiftSH (−d2) heap1))
                  l2
```

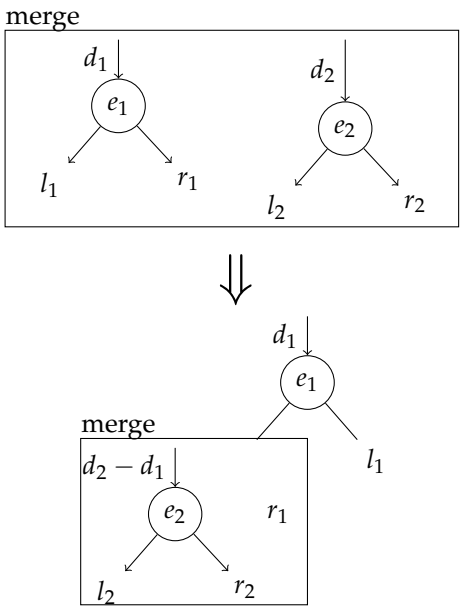


Figure 16: Heaps merge: case $d_1 < d_2$

We merge the trees following their rightmost paths, and swap the right and the left child in the resulting tree. This case is depicted in Figure 16. By this mean, the tree tends to grow "from the inside" rather than the rightmost path.

If the roots of the two tree we want to merge have the same delay, we do not increase the resulting tree size and we merges their events instead , otherwise we would have a 0 delay and violate our invariant. This case is depicted in Figure 17. The fact that we forbid null delay is reducing the tree size, but we could as well allow it and then treating this special case in the *headT* function rather than in the merge function. We can use *mergeSH* over infinite heaps because it is lazy on the childs of its parameters.

5 Complexity Analysis

In this section we will study space and time complexity of a syntactic implementation and the heaped implementation. We define the size of a tile by $n = n_e + n_d$ where n_e is the number of events and n_d is the number of delays in it.

5.1 Syntactic Complexity

A syntactic implementation suffer from two algorithmic problems, about time and space complexity.

Firstly, there is no mechanism to balance the structure built. The complexity of the in-time rendering depends on the way the tile has been constructed. For instance, a call to *firstE* is linear on a tile that has been constructed starting "from the end", because the first events are in the bottom of the syntactic structure. In general, all operation are in $\mathcal{O}(n)$, where n is the size of the tile.

Secondly, given a tile t of any event type, there is no bound to the size of a tile t' such that $t \equiv t'$. We can have an arbitrary number of delays because of the property $delay\ (a + b) \equiv delay\ a\ \%\ delay\ b$. With the implementation described in 3.1, the space complexity is $\mathcal{O}(n_e + n_d)$.

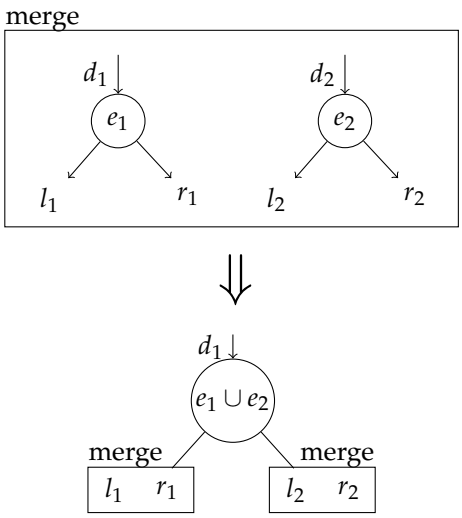


Figure 17: Heaps merge: case $d_1 = d_2$

5.2 Heaped Complexity

With the heaped implementation and assuming that *Duration* have $\mathcal{O}(1)$ space complexity (for instance when a bounded number of delay values are used), the space complexity of a tile with n_e events and n_d delays is $\mathcal{O}(n_e)$ since the invariant of the heap imply $n_d = \mathcal{O}(n_e)$.

It can be proven [7] that *mergeSH* have an amortized complexity of $\mathcal{O}(\log n)$. Hence, both % and *tailT* have a logarithmic time complexity and *headT* have a constant time complexity.

6 Performances

In this section, we will compare the *Twisted* implementation [5], and the one introduced in this paper. In all the following measures, we simulate the rendering of a tile by recursively extracting all the first events of the tile. We use GHC 7.8.4 with no runtime tuning on an AMD Opteron 6174 (2.2 GHz). More specifically, we will benchmark the normalization of tiles defined as follow:

$$
\begin{aligned}
&\mathit{tileL}\ 0 = \mathit{delay}\ 0\\
&\mathit{tileL}\ n = (\mathit{delay}\ 1 \,\%\, \mathit{event}\ ()) \,\%\, \mathit{tileL}\ (\mathit{pred}\ n)\\
&\mathit{tileR}\ 0 = \mathit{delay}\ 0\\
&\mathit{tileR}\ n = \mathit{tileR}\ (\mathit{pred}\ n) \,\%\, (\mathit{delay}\ 1 \,\%\, \mathit{event}\ ())\\
&\mathit{parallelTile}\ t\ n = \mathit{foldl}\ (\%)\ (\mathit{delay}\ 0)\ (\mathit{replicate}\ n\ (\mathit{re}\ t))\\
&\mathit{tileLeft} = \mathit{parallelTile}\ (\mathit{tileL}\ 2000)\ 16\\
&\mathit{tileRight} = \mathit{parallelTile}\ (\mathit{tileR}\ 2000)\ 16
\end{aligned}
$$

The performance of the normalization of a tile is very sensitive to its expression with the syntactic representation. The *tileR* function generate a worst case for syntactic representation (*parallelTile* do not).

Due to the space complexity described in section 5.1, the extraction of the first events of a tile does not guaranty that the remainder is smaller than the original tile. The implementation described in [5] suffer from this, the number of delays in $\mathit{tailT}\ t$ increases. This is the reason why in Figure 18 the reduction time of $\mathit{firstE}\ (\mathit{tailT}^n t)$ is proportional to n.

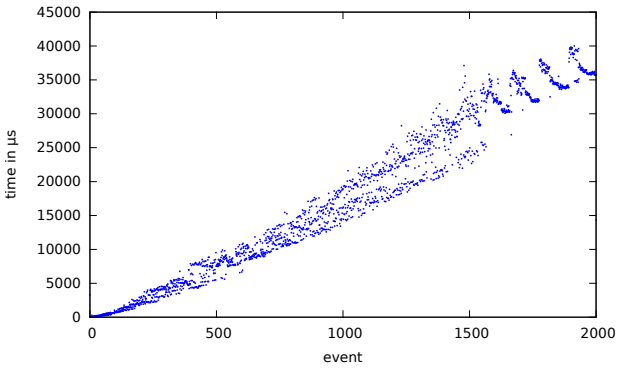


Figure 18: Normalization of *tileLeft* with the *Twisted* implementation

Thanks to the property $\mathit{delay}\ a \,\%\, \mathit{delay}\ b \equiv \mathit{delay}\ (a + b)$, we can always merge two consecutive delays and then reduce the tile size. By modifying the implementation of the % operator in *Twisted* to guaranty that we never have two consecutive delays, we have a constant time call to *firstE*. In Figure 19 we see that the patched *Twisted* implementation can perform an in-time on-the-fly rendering on

a left parenthesized tile. The very first call to *firstE* is longer because it trigger the evaluation of the tile.

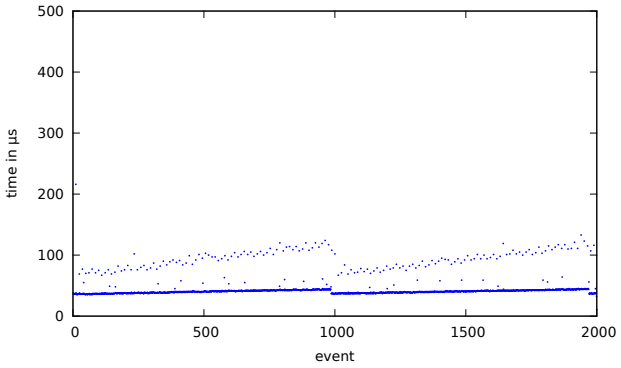


Figure 19: Normalization of *tileLeft* with the patched *Twisted* implementation

But the *Twisted* implementation stay very sensitive to the expression of the tile. In Figure 20 we see the normalization of the same tile but right parenthesized. The first events are the deepest leafs of an imbalanced syntactic tree, hence the reduction time of *firstE* is proportional to the size of the tile.

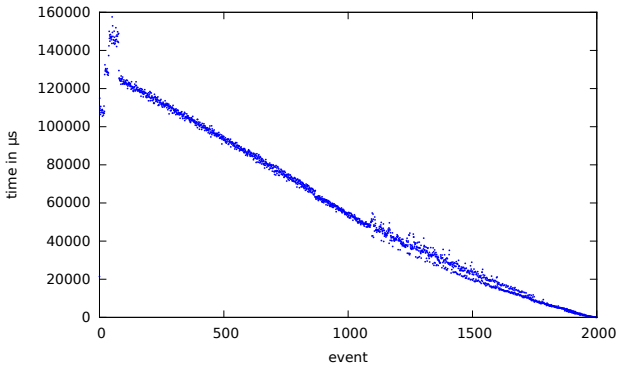


Figure 20: Normalization of *tileRight* with the patched *Twisted* implementation

The performance of the *Heaped* implementation performing the normalization of the same tile as in Figure 20 (a tile constructed "backward") is shown in Figure 21. The average call to *firstE* is less than 50 μs. The garbage collector periodically cause longer execution time. Because of the complicated complexity analysis of the skew heaps and the impact of the lazy semantics, a worst case for the *Heaped* implementation is yet to find.

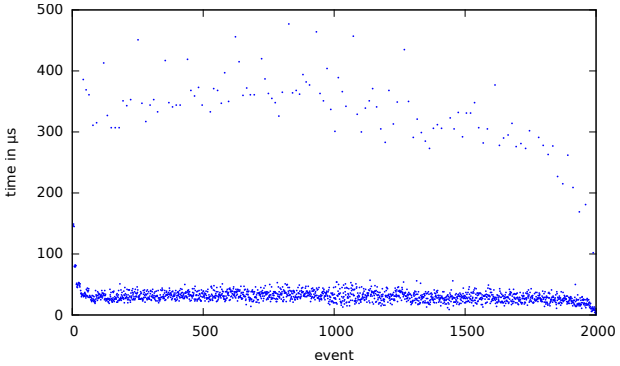


Figure 21: Normalization of *tileRight* with the new *Heaped* implementation (some few points are between 500 and 5000 μs)

The normalization of a left parenthesized tile with the *Heaped* implementation is depicted in Figure 22. The first call to *firtsE* last 1400 μs, the average is 6 μs.

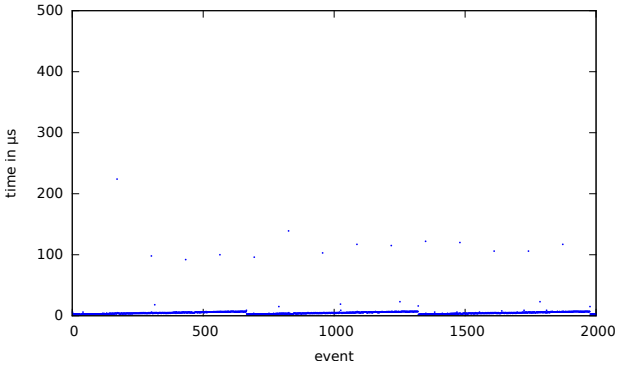


Figure 22: Normalization of *tileLeft* with the new *Heaped* implementation (some few points are between 500 and 1500 μs)

7 Infinite Tiles

As the lazy semantics of Haskell allows the programmer to write infinite data structures such as lists, we would like to be able to write infinite tiles. In this section we study the possibility of representing and playing an infinite tile. Such a tile that may results from an equation of the form $x = m \,\%\, re\ x$ [4]. Because we are interested in relevant tile from a musical point on view, we will consider only the tiles with a finite number of events at each time, and where all delay d_i between consecutive set of events verify for all i: $0 < m \leqslant d_i \leqslant M$ for some m and M. This mean in particular that we can not have an infinite number of events in a finite time interval.

7.1 Syntactic Implementation

With the syntactic representation, we observe that it is impossible to build such an infinite tile with a finite duration. Because the duration of the tile is the sum all the delays that composes it, and a infinite tile have an infinite number of positive delays, we would need to add an infinite number of negative delays to make the sum finite. We can write a function that compute an potentially infinite tile from a potentially infinite list for instance, but it will suffer from a memory leak because of the infinite number of negative delays accumulated needed to have a finite duration. The Figure 23 depict such a tile.

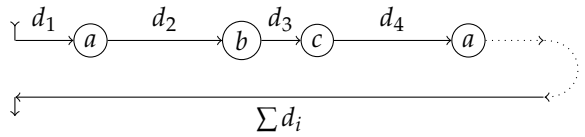


Figure 23: A potentially infinite zigzag tile

7.2 Heaped Implementation

As in tiled Euterpea, it is possible to represent an infinite tile with the *Heaped* implementation. The reason is that the set of *SHeap e* with empty right childs is isomorphic to the list type $[(Duration, MultiSet\ e)]$ where all the durations in it are positives except maybe the first one and all multisets are non empty. Hence, we can build infinite tiles given an infinite list.

We define a function *sToT* as Stream To Tile, that takes a potentially infinite list and returns an infinite tile of finite duration that encode this list. Such a tile is depicted in Figure 24.

```
sToT l = Tile 0 (lToSH l)
lToSH :: [(Duration, MultiSet e)] → SHeap e
lToSH [] = Empty
lToSH ((d, e) : (0, e') : l) = lToSH ((d, union e e') : l)
  -- we don't want o's
lToSH ((d, e) : l) = SHeap d e (lToSH l) Empty
```

The *lToSH* function computes a *SHeap* from the list while reducing o delays, and the *sToT* function returns a null duration Tile with this heap.

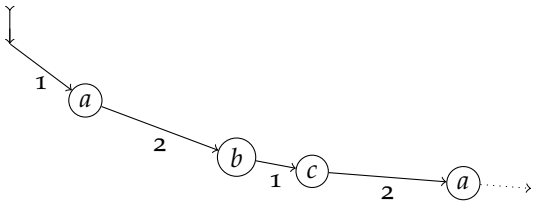


Figure 24: An infinite heaped tile

Unlike the implementation in 7.2, here we can build and render an infinite tile with no memory leaks.

8 Encoding PTM into TPTM

As a consequence of the possibility to compute infinite Heaped tile, we can write conversion function from infinite PTM to TPTM. This implementation suffers from a memory leak when the PTM converted is have an infinite duration.

We use this simple definition for PTM, and a *Event* to encode temporal media in term of events:

$$
\begin{aligned}
&\textbf{data } \mathit{Media}\ a = \mathit{Prim}\ \mathit{Duration}\ a \\
&\qquad\qquad \mid \mathit{Wait}\ \mathit{Duration} \\
&\qquad\qquad \mid \mathit{Media}\ a :+: \mathit{Media}\ a \\
&\qquad\qquad \mid \mathit{Media}\ a :=: \mathit{Media}\ a \\
&\textbf{data } \mathit{Event}\ a = \mathit{On}\ a \mid \mathit{Off}\ a\ \textbf{deriving } (\mathit{Eq}, \mathit{Ord})
\end{aligned}
$$

The constructors *Prim d a* represents a media *a* during a time *d*, *Wait d* is a delay, and the remaining constructors are pretty straightforward. In TPTM, we encode a media *a* for a duration *d* by two event separated by a time *d* marking the beginning and the end of *a*.
The tiles involved here have the property:

(P) All the events embedded are between the two synchronization marks: There is no event before the pre marker, and no event after the post marker.

A TPTM encoding a PTM is a well parenthesized sequence of event *On a* and *Off a*. We define a function *mToT* as Media To Tile that computes such a tile:

$$
\begin{aligned}
&\mathit{mToT} :: \mathit{Ord}\ a \Rightarrow \mathit{Media}\ a \to \mathit{Tile}\ (\mathit{Event}\ a) \\
&\mathit{mToT}\ (\mathit{Prim}\ d\ a) = \\
&\quad \mathit{event}\ (\mathit{On}\ a) \,\%\, \mathit{delay}\ d \,\%\, \mathit{event}\ (\mathit{Off}\ a) \\
&\mathit{mToT}\ (\mathit{Wait}\ d) = \mathit{delay}\ d \\
&\mathit{mToT}\ (m_1 :=: m_2) = \mathit{parM}\ (\mathit{mToT}\ m_1)\ (\mathit{mToT}\ m_2) \\
&\mathit{mToT}\ (m_1 :+: m_2) = \mathit{seqM}\ (\mathit{mToT}\ m_1)\ (\mathit{mToT}\ m_2)
\end{aligned}
$$

The *seqM* and *parM* functions are the tiled implementation of :+: and :=: respectively, in order to be able do deal with infinite tiles, these functions can not be implemented naively. The *seqM* function works under the hypothesis that both operands verify (P).

$$
\begin{aligned}
&\mathit{parM}\ (\mathit{Tile}\ d_1\ h1)\ (\mathit{Tile}\ d_2\ h2) = \\
&\quad \mathit{Tile}\ (\mathit{max}\ d_1\ d_2)\ (\mathit{mergeSH}\ h1\ h2) \\
&\mathit{seqM}\ (\mathit{Tile}\ d_1\ h1)\ \ \sim(\mathit{Tile}\ d_2\ h2) = \\
&\quad \mathit{Tile}\ (d_1 + d_2)\ (\mathit{lazyMergeSH}\ h1\ (\mathit{shiftSH}\ d_1\ h2))
\end{aligned}
$$

The implementation of *parM* is pretty straightforward, we notice that *parM* is lazy on durations, and *mergeSH* can handle infinite heaps, so such a function suit its purpose. However, the implementation of *seqM* is less obvious because it have to be lazy on its second parameter in order to be able to handle an infinite PTM such as $m_1 :+: (m_2 :+: (m_3 :+: (m_4 ....$

The first unwanted strictness come from the fact that in Haskell, pattern matching in case expression are strict by default. Without the "lazy-pattern" tilde, when *seqM a b* is evaluated, it will trigger the evaluation of *a* and *b*, *b* can also be of the form *seqM c d*, hence, it will trigger an infinite computation. Such a pattern matching cannot fail because *Tile e* have a single constructor. We could as well made the program lazy on the second operand by other mean, but for the sake of simplicity, we use the $\sim(\mathit{pattern})$ syntax.

The second unwanted strictness comes from *mergeSH* because it needs to evaluate both its operand in order to see from which one the first event of the resulting tree comes from. We use the hypothesis (P) to implement a lazier merge.

$$
\begin{aligned}
&\mathit{lazyMergeSH} :: \mathit{SHeap}\ e \to \mathit{SHeap}\ e \to \mathit{SHeap}\ e \\
&\mathit{lazyMergeSH}\ \mathit{Empty}\ h = h \\
&\mathit{lazyMergeSH}\ (\mathit{SHeap}\ d\ e\ l\ r)\ h = \\
&\quad \mathit{SHeap}\ d\ e\ (\mathit{lazyMergeSH}\ r\ (\mathit{shiftSH}\ (-d)\ h))\ l
\end{aligned}
$$

The *lazyMergeSH* function is lazy on the second parameter and works only under the hypothesis that the first event of the second heap is after the last event of the first heap. The algorithm is exactly the same as *mergeSH*, but by hypothesis we are always in the *LT* case.

The *mToT* function build a tile of the duration of the media, such a duration can be undefined in case of infinite PTM such as *m* in $m = m_1 :+: m$. It follows that one will prefer the use of the function *mToReT* which returns the reset of that tile, of duration 0.

$$
\begin{aligned}
&\mathit{mToReT} :: \mathit{Ord}\ a \Rightarrow \mathit{Media}\ a \to \mathit{Tile}\ (\mathit{Event}\ a) \\
&\mathit{mToReT} = \mathit{re} \circ \mathit{mToT}
\end{aligned}
$$

Such an implementation suffers from a memory leak if the PTM converted is infinite. Because the

function *mToT* will compute without evaluating the duration of the resulting tile, which can be infinite.

9 Rendering

The *firstE*, *firstD* and *tailT* performs an abstract rendering. In order to make more concretes experiments around TPTM, two players have been written. The first one is a MIDI player, and the second one an audio player.

These two players relies on the sound server JACK that provides a real-time engine and primitive to send both MIDI and audio. The input of JACK is bufferized, this conveniently loosen the real-time constrains imposed to the Haskell runtime.

9.1 MIDI Player

The MIDI player takes as input a tile of MIDI messages and feed the JACK server by performing an on-the-fly normalization of the tile. The *firstE* function performs an agnostic accumulation of the MIDI events, the player relies on downstream components to compute the semantic of simultaneous events.

This player have been used to play standard MIDI files. Such a file can be seen as a list of parallel tracks (usually one per instrument), where a track is a sequence of delays and MIDI events. Hence, we encode a MIDI file by a tile of the form $re\ t_1 \,\%\, re\ t_2 \,\%\, re\ t_3 \% \cdots$ where each t_n is a positive tile encoding of the nth MIDI track.

9.2 Audio Player

We implemented a player that is capable of reading and playing audio tile as well. A digital audio stream is a continuous signal discretized at a constant framerate. For performance reasons, rendering a tile with one event per audio frame was not feasible. Thus, an audio tile is based on the granular synthesis technique. The audio events are "grain", some few milliseconds of audio stream. These grains overlap and are placed in an envelope in order to avoid artifacts.

Thanks to this technique, we can easily change the tempo without modifying the pitch, as it suffice to modify the delays in tile separating the grains. However, significant speed change generate a lot of artifacts.

A reduction have to be performed on the events returned by *firstE*, from a multiset of grains we compute a vector of audio frames.

These experiments open the way to musical experimentation. We are currently working on a player that can simultaneously read both MIDI and audio.

10 Conclusion

The implementation proposed in this paper have good algorithmic properties and can handle infinite temporal media, just as Tiled Euterpea could [4].

In further research we would like to extend TPTM model in order to handle interactive temporal media. In particular, we are interested in being able to describe some synchronizations between the rendering of a tile and an external stream of events (the music played by a musician for instance). We are also interested in expressing complex interactive programs with conditional renderings.

The tree structure of our implementation seems to be adapted to this extension as it induces a partial order between the events that is compatible with the temporality of those events. This tree describes a non-determinism between the order of the events to be played that is resolved by the normalization. An arbitrary arity tree could express a non-determinism between the order of external events that would be resolved by the monitoring of external events. Of course, this would imply exploring the possible links with other known approaches such as, in particular, Functional Reactive Programming approach.

References

[1] T. Bazin and D. Janin. Flux média tuilés polymorphes: une sémantique opérationnelle en

Haskell. 2015.

[2] P. Hudak. An algebraic theory of polymorphic temporal media. In *Proceedings of PADL'04: 6th International Workshop on Practical Aspects of Declarative Languages*, pages 1–15, 2004.

[3] P. Hudak. *The Haskell School of Music : From signals to Symphonies*. Yale University, Department of Computer Science, 2013.

[4] P. Hudak and D. Janin. Tiled polymorphic temporal media. In *ACM Workshop on Functional Art, Music, Modeling and Design (FARM)*, pages 49–60. ACM Press, 2014.

[5] P. Hudak and D. Janin. From out-of-time design to in-time production of temporal media. Technical report, LaBRI, Université de Bordeaux, 2015.

[6] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3):465–483, May 1996.

[7] D. D. Sleator and R. E. Tarjan. Self adjusting heaps. *SIAM J. Comput.*, 15(1).